

# Ch MPI: Interpretive Parallel Computing in C

*The message passing interface lets users develop portable message passing programs for parallel computing in C, C++, and Fortran. When combined with an MPI C/C++ library, Ch, an embeddable C/C++ interpreter for executing C/C++ programs interpretively, lets developers rapidly prototype MPI C/C++ programs without having to compile and link.*

**T**he message-passing interface defines a set of API functions that let users write high-performance parallel programs to exchange data between processes and complete tasks in parallel.<sup>1</sup> The MPI standard was designed to support portability and platform independence. Therefore, an application source code written with MPI is portable among different platforms. Traditionally, developers achieve high-performance parallel computing using a compiled language such as C or Fortran. However, a parallel scientific and engineering application typically spends most of its time on a small amount of time-critical code. The rest of the code typically deals with memory management, error handling, I/O, user interaction, and parallel process startup. Given this and the need for rapid algorithm prototyping, the scientific community is increasingly using scripting languages. Developers can use such languages to rapidly implement, test, and debug parallel algorithms without needing to recompile and link every time they make a modification. By avoiding the compile/link step,

scripting languages help developers deal with the error-prone or high-level portions of parallel applications. Once the testing and debugging phases are completed, they can recode the parallel scripts in a compiled language—such as C or Fortran—for optimized performance.

Researchers have created MPI bindings for several scripting languages, including MPI for Python,<sup>2,3</sup> MatlabMPI,<sup>4</sup> MPI Toolbox for Matlab, MPI Toolbox for Octave, MPI Ruby,<sup>5</sup> and Parallel::MPI for Perl. A language-MPI binding is an API that lets users develop MPI programs in that language. Therefore, these bindings offer all the benefits of a high-level scripting language within a parallel runtime environment. However, some of them have incomplete MPI functionality. Because scripting language syntax differs from C, C++, or Fortran, the potential conversion from scripting MPI programs to compiled MPI programs is complicated. Some scripting languages also require developers to purchase a license for each node to install the necessary software. Further, despite scripting languages' many benefits, researchers working on a computationally intensive parallel application are most likely writing in C, C++, or Fortran. For people interested in learning parallel programming, C is also a good choice due to its broad use and international standard. The new numerical features in C99, the latest C standard, greatly improve the performance of C programs. Given these factors, it would be beneficial to have a C-based scripting environment with MPI capabilities.

1521-9615/10/\$26.00 © 2010 IEEE  
COPUBLISHED BY THE IEEE CS AND THE AIP

YU-CHENG CHOU

*Chung Yuan Christian University, Taiwan*

STEPHEN S. NESTINGER

*Worcester Polytechnic Institute*

HARRY H. CHENG

*University of California, Davis*

Here, we introduce a generalized method for creating MPI bindings for Harry Cheng's Ch ([www.softintegration.com](http://www.softintegration.com)), an embeddable C/C++ interpreter, based on different MPI implementations.<sup>6</sup> We use the MPICH2 C library ([www-unix.mcs.anl.gov/mpi/mpich2](http://www-unix.mcs.anl.gov/mpi/mpich2)) as an example to illustrate the creation of a Ch MPI package (<http://iel.ucdavis.edu/projects/chmpi>), which interfaces all of the library's MPI functions. The Ch MPI website also contains a Ch binding for the local-area multicomputer/MPI (LAM/MPI) C library. Using our Ch MPI package, developers can treat a C MPI program as a Ch script and run it directly within the Ch environment across different platforms, without needing to compile and link the program to create platform-dependent executables.

### Ch: A C/C++ Script Computing Environment

Ch is an embeddable C/C++ interpreter for cross-platform scripting, shell programming, numerical computing, network computing, and embedded scripting.<sup>6</sup> Ch is an extension and enhancement of the most popular Unix/Windows C computing environment. Because it's platform independent, a Ch program developed on one platform can be executed on a different platform without recompiling and linking. Ch supports all features of the ISO C90 standard. Many Ch features—such as complex numbers and variable-length array (VLA)—have been added to C99. Ch also supports computational arrays, as in Fortran 90 and MATLAB, for linear algebra and matrix computations (we offer an example later that uses Ch computational arrays to solve a simple matrix equation).

As a scripting environment, Ch allows for quick and easy application development and maintenance. Developers can use it to glue together different modules and tools for rapid application prototyping. Ch users can directly invoke system utilities that would otherwise require many lines of code to implement. Once they've completed the rapid prototyping phase, developers can replace a Ch script's commands and utilities with equivalent C code. The code can also be interpretively run in Ch. This lets developers rapidly test applications before they're compiled, thus saving development time and effort. Once the tests are complete, users can readily compile their code for optimized performance. As a C/C++ interpreter with a user-friendly integrated development environment called ChIDE, Ch is especially suitable for teaching introductory computer programming (<http://iel.ucdavis.edu/cfores>).<sup>7</sup> Ch is also suitable for Web-based client-server computing.

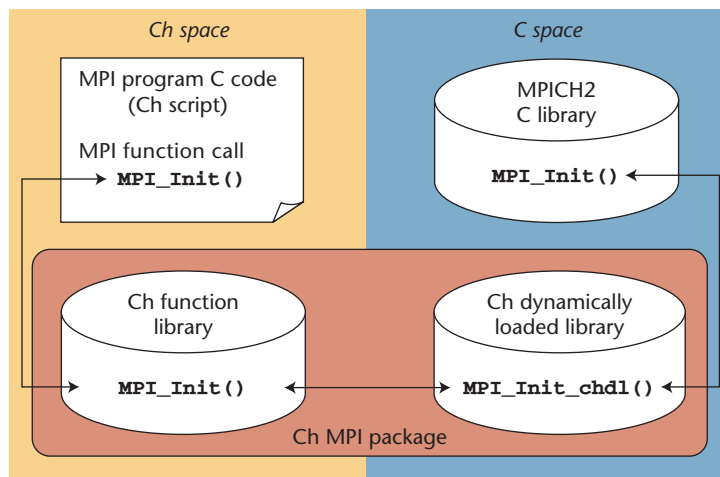


Figure 1. Interfacing a Ch script with the MPICH2 C library. A Ch MPI package lets users call MPICH2 C library functions from within a Ch script.

Ch programs can be used through a Common Gateway Interface (CGI) in a Web server for Web-based computing and through Mobile-C ([www.mobilec.org](http://www.mobilec.org)), a mobile-agent platform, for mobile-agent based computing.

### Ch MPI

Our interfacing method applies to any C MPI implementation; here, we use the MPICH2 C library example to illustrate how to create a Ch MPI package for running an MPI program, or *Ch script*, in an interpretive environment.

#### Interfacing with an MPI C Library

Ch MPI is a Ch binding to the MPICH2 C library that makes all the MPICH2 C library functions callable from a Ch script. The Ch software development kit (SDK) allows for porting the MPICH2 C library into the Ch space by creating a Ch MPI package that binds to the MPICH2 C library. As Figure 1 shows, a Ch MPI package contains a Ch function library in the Ch space and a Ch dynamically loaded library in the C space. These two libraries provide wrapper functions that let us call MPICH2 C library functions from within the Ch space. For an MPICH2 C library function, a wrapper function consists of a `chf` function from the Ch function library and a `chdl` C function from the Ch dynamically loaded library. We implement each `chf` function in a separate file with the `.chf` file extension. All the `.chf` files constitute the Ch function library. In contrast, we implement all `chdl` C functions in a C program that is compiled to generate the Ch dynamically loaded library, which is loaded when an MPI program executes.

```

#ifdef _CH_
#pragma package <chmpi>
#include <chdl.h>
LOAD_CHDL(mpi);
#endif

int MPI_Init(int *argc, char ***argv) {
    void *fptr;
    int retval;

    fptr = dlsym(_Chmpi_handle, "MPI_Init_chdl");
    if(fptr == NULL) {
        fprintf(stderr, "Error: %s(): dlsym(): %s\n", __func__, dlerror());
        return -1;
    }
    dlrnfun(fptr, &retval, MPI_Init, argc, argv);
    return retval;
}

```

Figure 2. The `chf` function `MPI_Init()` is implemented as a separate function file, `MPI_Init.chf`, in a Ch MPI package. Such function files constitute the Ch function library of a Ch MPI package. The `chf` function `MPI_Init()` calls the `chdl` C function `MPI_Init_chdl()`.

Figure 1 illustrates the underlying Ch MPI concept: an MPICH2 C library function, `MPI_Init()`, is called by a Ch script—that is, a C program with MPI function calls—in the Ch space. To invoke the C function `MPI_Init()`, Ch searches the Ch function library for the corresponding Ch function `MPI_Init()` implemented in the file `MPI_Init.chf` and passes parameters to the function. The Ch function `MPI_Init()` then searches through the Ch dynamically loaded library for the corresponding `chdl` C function, `MPI_Init_chdl()`, and passes parameters to it. The `MPI_Init_chdl()` function then invokes the function `MPI_Init()` in the MPICH2 C library. Any return values are passed back to the initial function call in the Ch space.

We can readily run a C MPI program in Ch by typing in the script's name at a Ch shell's command prompt. This lets us quickly modify program code and generate results without recompiling and linking the program. When a C MPI program is still under development, the error messages given by interpretively executing the program code in Ch are generally more informative than the messages given by compiling the program using conventional C compilers. For example, this occurs in situations where an MPI program contains syntax errors or accesses elements beyond an array's extent.

The key components of a Ch MPI package include the header file `mpi.h`, `chf` functions, and `chdl` C functions. To illustrate them, we'll use the header file `mpi.h`, the `chf` function `MPI_Init()`, and the `chdl` C function `MPI_Init_chdl()`.

**Header file.** The header file `mpi.h` is included at the beginning of every C MPI program. This header file is the same as the one in the MPICH2 C library, with the addition of several statements. The macro `LOAD_CHDL(mpi)` invokes a function named `dlopen()` that loads the Ch dynamically loaded library `libmpi.dl` into the Ch interpretive environment. The function `dlopen()` returns a handle `_Chmpi_handle`, which is used in subsequent calls to functions `dlsym()` and `dlclose()`. The same macro invokes the function `dlclose()` to release the dynamically loaded library `libmpi.dl` from the Ch interpretive environment when a C MPI program execution completes.

**The `chf` Ch function.** As Figure 2 shows, the `chf` function `MPI_Init()` is implemented in the file `MPI_Init.chf`. The function call `fptr = dlsym(_Chmpi_handle, "MPI_Init_chdl")` returns the address of the symbol `MPI_Init_chdl` inside the dynamically loaded library `libmpi.dl` pointed to by the handle `_Chmpi_handle`. The function call `dlrunfun(fptr, &retval, MPI_Init, argc, argv)` runs the `chdl` C function `MPI_Init_chdl()` found in the dynamically loaded library `libmpi.dl` through the address pointed to by the pointer `fptr`. The second parameter is the address of a variable `retval` containing the return value of the `chdl` C function `MPI_Init_chdl()`. Because, in this case, the third parameter, `MPI_Init`, is the name of the `chf` function itself, the number and types of the remaining parameters are first examined according to the function prototype of `MPI_Init()`.

The last two parameters, `argc` and `argv`, are then passed to the `chdl` C function `MPI_Init_chdl()` if the examination succeeds.

**The `chdl` C function.** Figure 3 shows the `chdl` C function `MPI_Init_chdl()`. Even though the `chf` function `MPI_Init()` passes two parameters to the `chdl` C function `MPI_Init_chdl()`, the latter function still only takes one parameter, `varg`, which is a void pointer to the parameter list. The function call `Ch_VaStart(interp, ap, varg)` obtains `interp`, an instance for the Ch interpreter, and initializes `ap`, an object for the parameter list. These two parameters will be used in subsequent calls to functions `Ch_VaArg()` and `Ch_VaEnd()`. The two consecutive calls to the function `Ch_VaArg()` save the values passed from the `chf` function `MPI_Init()` in the two parameters `argc` and `argv`, respectively. The function call `retval = MPI_Init(argc, argv)` invokes the function `MPI_Init()` in the MPICH2 C library and saves the return value in the variable `retval`. The function call `Ch_VaEnd(interp, ap)` releases the memory associated with the instance `interp` and object `ap`.

### Executing an MPI Program across Different Platforms

To illustrate how a Ch MPI package allows for interpretive parallel computation, we use an MPI program, `cp_i.c`,<sup>8</sup> for calculating  $\pi$  (see Figure 4). With a Ch MPI package, we can execute the `cp_i.c` program in Ch as is, without any modification. The program repeatedly asks users to input the number of intervals to calculate  $\pi$  and exits when the input value is 0. As Figure 4 shows, the `cp_i.c` program is based on the equation

$$\int_0^1 f(x)dx = \int_0^1 \frac{4}{1+x^2} dx = \pi.$$

Figure 5 shows an example of the numerical integration in the `cp_i.c` program (Figure 4). In this case, we use two processes, P0 and P1, to compute  $\pi$  with four intervals between 0 and 1. The approximate value of  $\pi$  is the summation of four rectangles' areas. Each rectangle has the same base  $b = 0.25$ . As Figure 5 shows, we obtain each rectangle's height as the value of  $f(x)$ , with  $x$  equal to the base's middle point. Also, each process computes the area of every other rectangle. Increasing the number of intervals between 0 and 1 improves the  $\pi$  value's precision, but magnifies the processes' workload.

```
EXPORTCH int MPI_Init_chdl(void *varg) {
    ChInterp_t interp;
    ChVaList_t ap;
    int *argc;
    char ***argv;
    int retval;

    Ch_VaStart(interp, ap, varg);
    argc = Ch_VaArg(interp, ap, int *);
    argv = Ch_VaArg(interp, ap, char ***);
    retval = MPI_Init(argc, argv);
    Ch_VaEnd(interp, ap);
    return retval;
}
```

Figure 3. The `chdl` C function `MPI_Init_chdl()` in a Ch MPI package. Such `chdl` C functions are implemented in one single file that is compiled to generate the Ch dynamically loaded library of a Ch MPI package. The `chdl` C function `MPI_Init_chdl()` calls the function `MPI_Init()` in the MPICH2 C library.

After a few variable declarations, the `cp_i.c` program calls the function `MPI_Init()` to initialize the MPI environment. The function `MPI_Init()` is required in every MPI program and must be the first MPI function called. The function call `MPI_Comm_size()` saves the number of processes the user has started in the parameter `numprocs`. The function call `MPI_Comm_rank()` saves the rank of a process in the parameter `myid`. In the MPI, a communicator encompasses a group of processes that can communicate with each other. All MPI messages must specify a communicator. Here, the `cp_i.c` program uses the default communicator `MPI_COMM_WORLD`, which contains all processes. Next, the user inputs the value of  $n$  (the number of intervals) to the master process, whose `myid` is 0. The function call `MPI_Bcast()` sends the  $n$  value from the master process to all the worker processes, whose `myid` is not 0. After the function call `MPI_Bcast()`, all processes have  $n$  and their own `myid`, which is enough information for each process to compute its `mypi`. Each process does this by calculating the total area for a certain number of rectangles, whose ordinal numbers begin with `myid+1` and increase by `numprocs`. Next, the function call `MPI_Reduce()` sums up all `mypis` from all processes. The result is saved in the variable `pi` of the master process. All processes then return to the top of the loop, except for the master process, which first prints the results. When the user types "0" in response to the interval number request, the loop terminates and all processes call the function `MPI_Finalize()`, which terminates the MPI environment.

```

/*****
* File: cpi.c
* Purpose: Parallel computation of pi with the
*          number of intervals from standard input.
*****/
#include <mpi.h>
#include <stdio.h>
#include <math.h>

int main(int argc, char *argv[] ) {
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    while(1) {
        if(myid == 0) {
            printf("Enter the number of intervals:
                    (0 quits) ");
            scanf("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if(n == 0) { // program quits if the interval
                    // is 0
            break;
        }
        else {
            h = 1.0 / (double)n;
            sum = 0.0;
            for(i=myid+1; i<=n; i+=numprocs) {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM,
                    0, MPI_COMM_WORLD);
            if(myid == 0) {
                printf("PI is %.16f, Error is %.16f\n",
                    pi, fabs(pi - PI25DT));
            }
        }
    }
    MPI_Finalize();
    return 0;
}

```

Figure 4. The `cpi.c` program for the parallel computation of  $\pi$ . The program repeatedly asks users to input the number of intervals to calculate  $\pi$ , and exits when the input value is 0.

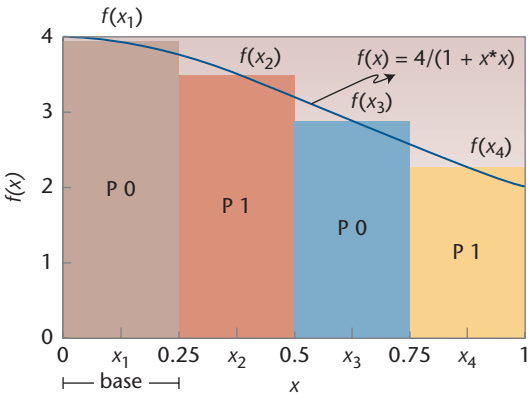


Figure 5. The `cpi.c` program’s principle for computing  $\pi$  with multiple processes. Here, the approximate value of  $\pi$  is the summation of four rectangles’ areas calculated by two processes.

Figure 6 shows the `cpi.c` script execution sequence in Ch using the MPICH2-based Ch MPI package in an environment with two different platforms—Linux and Windows—which both contain the script `cpi.c`. MPICH2 provides the super multipurpose daemon (SMPD), which supports an MPI program’s execution on a combination of Linux and Windows platforms. The sequence in Figure 6 includes four steps:

- The user starts the SMPDs on both machines.
- The user runs the `mpiexec` program to send the execution message (`mpiexec` command line arguments) to the local SMPD.
- The local SMPD distributes the message to the remote SMPD.
- According to the received execution message, each SMPD launches a Ch interpreter to execute the specified C MPI script, `cpi.c`, through the Ch MPI package.

Figure 7 shows the master process output from executing `cpi.c` on both Linux (phoenix) and Windows (mouse1) platforms. The master process is on the local phoenix machine. The machines argument in Figures 6 and 7 is an MPICH2 host file used to specify the information for all the machines contributing to the parallel computation. Likewise, the `password` argument (also shown in both figures) is a password file used to specify a username and password. Each machine invokes a Ch interpreter by placing `ch` as the executable in the `mpiexec` command line. The `-u` is a `ch` option that unbuffers the standard output stream, which is line buffered by default. When line buffered, the system waits for a new line character before writing the buffer contents onto the terminal screen. Using the `-u` option can prevent the `cpi.c` execution from halting.

### Performance Evaluation of Ch MPI

Here, we present results from comparing C MPI’s bandwidth and latency with those of Ch MPI. We also show execution performance for Ch MPI and MatlabMPI using programs for parallel and serial calculation of  $\pi$ , and for C MPI and Ch MPI using a program for 2D matrix multiplication.

### Performance Evaluation Setup

Our 64-Bit Linux cluster contains 10 nodes, each with 8 Gbytes of RAM and eight quad-core AMD Opteron processors running at 1.8 GHz. All nodes are connected through a switch with a 1 Gigabit/sec transmission rate. The heterogeneous network cluster contains Linux and Windows

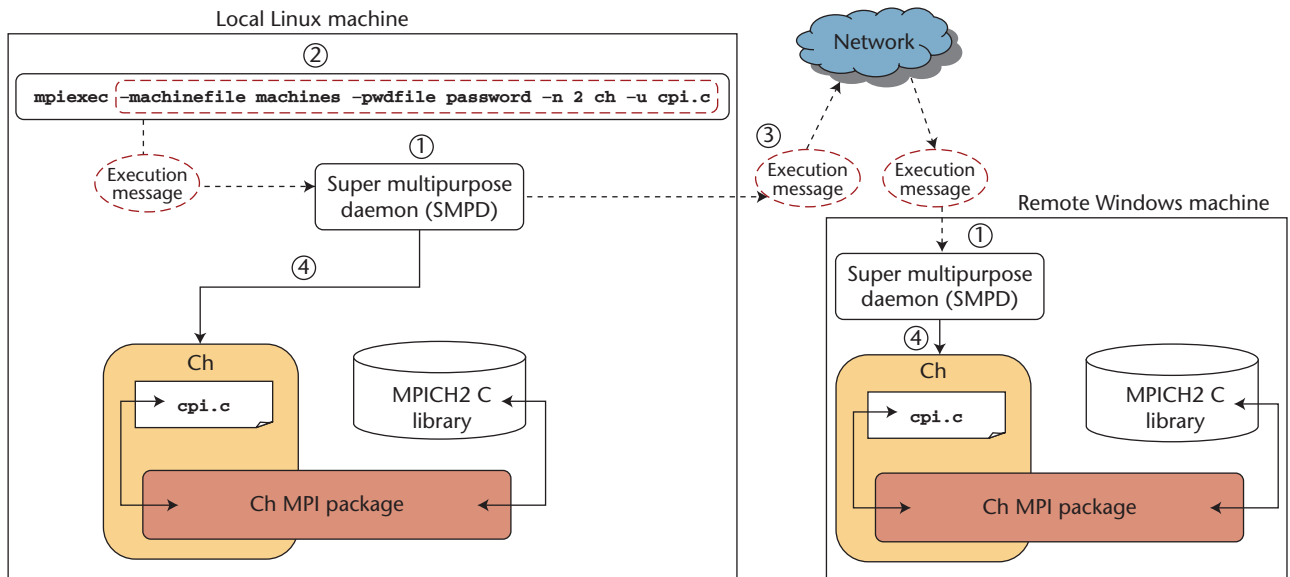


Figure 6. Executing `cpi.c` across Linux and Windows platforms in Ch using the MPICH2-based Ch MPI package. Each SMPD launches a Ch interpreter to run `cpi.c` to calculate a partial  $\pi$ . Through Ch and Ch MPI, we can perform parallel computation across heterogeneous platforms with identical Ch scripts.

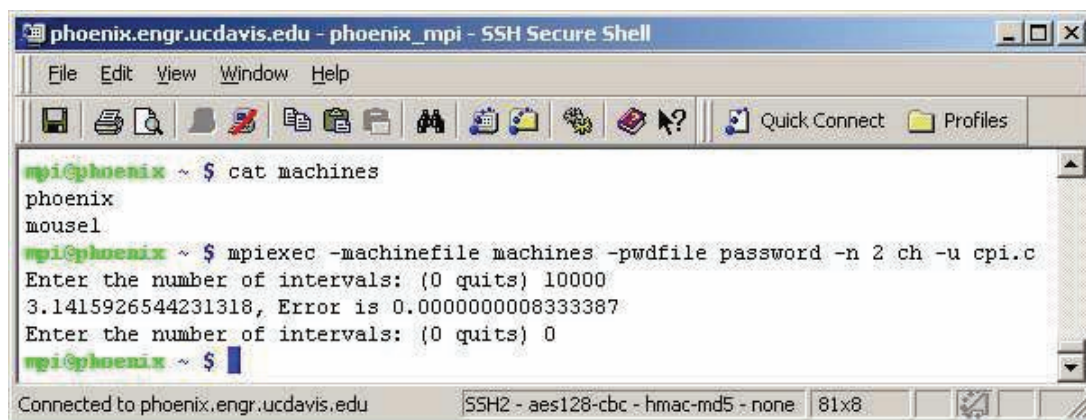


Figure 7. The master process output when executing `cpi.c` on Linux (phoenix) and Windows (mouse1) platforms. The master process is on the local phoenix machine, while the worker process is on the remote mouse1 machine.

machines. Each machine has an Intel Pentium 4 processor running at 3.2 GHz and 512 Mbytes of RAM. All of the machines are connected through a switch with a 100 Megabit/sec transmission rate.

We also use two homogeneous network clusters: one consists of only Linux machines, and the other of only Windows machines. Each machine in both clusters has one Intel Pentium 4 processor running at 3.2 GHz and 512 Mbytes of RAM. All of the machines are connected through a switch with a 100 Megabit/sec transmission rate.

### Bandwidth and Latency Benchmark

We benchmarked and compared Ch MPI to a native C MPI implementation, MPICH2,

in terms of bandwidth and latency. Hardware setups for the benchmark were as follows (the letters correspond to the results in Figures 8 and 9):

- two nodes from the 64-bit Linux cluster, using one processor from each node (a);
- one node from the 64-bit Linux cluster, using two of the node's processors (b);
- one Linux machine and one Windows machine from the heterogeneous network cluster (c);
- two Linux machines from the homogeneous Linux network cluster (d); and
- two Windows machines from the homogeneous Windows network cluster (e).

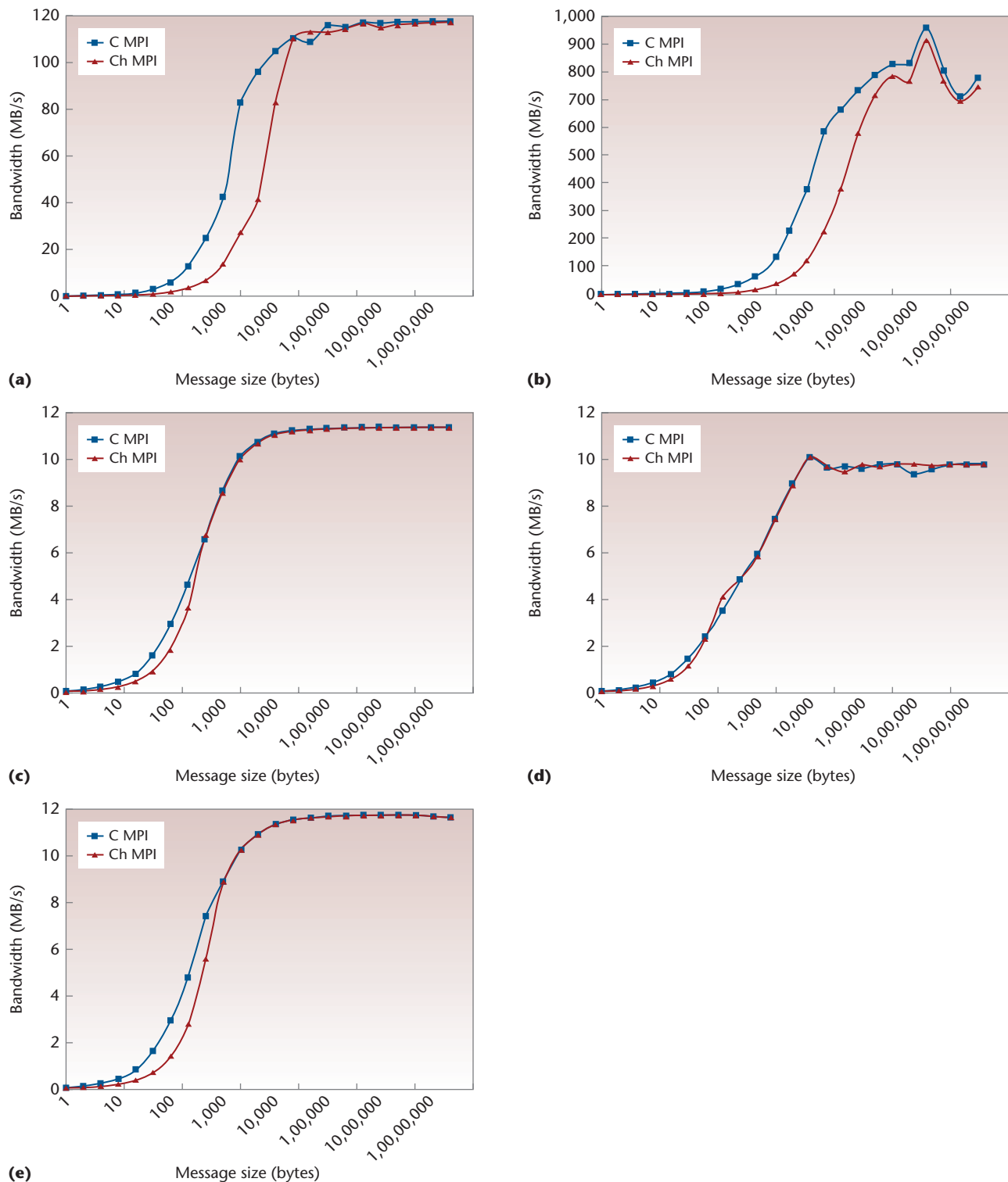


Figure 8. Bandwidth test results for C MPI and Ch MPI in semi-log plots. The hardware setups were (a) a 64-bit Linux cluster with two processors and two nodes, (b) a 64-bit Linux cluster with two processors and one node, (c) a heterogeneous network cluster, (d) a Linux network cluster, and (e) a Windows network cluster.

We obtained the benchmark programs running in both Ch MPI and native C MPI from Ohio State University's *MVAPICH* project (that is, an MPI implementation over the InfiniBand VAPI interface based on the MPICH implementation; see <http://mvapich.cse.ohio-state.edu>); they're also available on our own Web site (<http://iel.ucdavis.edu/projects/chmpi>). Both tests included the use

of a sender and receiver process, running on separate processors.

**Bandwidth testing.** In this test, we calculated bandwidths for different message sizes. For each message size, the sender process consecutively sent out a fixed number of messages of that size to the receiver process and waited for the receiver process to reply, which it would do only after receiving all of the messages. This procedure repeated for several iterations; we then calculated the bandwidth for each message size. We based the calculation on the number of bytes sent by the sender process and the elapsed time between when the sender process sent out the first message and when it received the reply from the receiver process. We used nonblocking MPI functions, `MPI_Isend()` and `MPI_Irecv()`, in the bandwidth test.

Figure 8 shows the bandwidth test results of C MPI and Ch MPI, with the  $x$ -axis on a base-10 logarithmic scale. Ch MPI's bandwidth is comparable to that of C MPI for cases (a) and (b) in the 64-bit Linux cluster setup. Ch MPI's bandwidth is very close to that of C MPI for cases (c) through (e) on the heterogeneous and homogeneous network cluster setups in both Linux and Windows.

**Latency testing.** In this test, we calculated latencies with respect to different message sizes. For each message size, the sender process sent out one message of that size to the receiver process and waited for the receiver process to reply. The receiver process received the sender's message and sent back a reply of the same message size. This procedure repeated for several iterations; we then calculated the average one-way latency regarding that message size. We used blocking MPI functions, `MPI_Send()` and `MPI_Recv()`, in the latency test.

Figure 9 shows the latency test results for C MPI and Ch MPI with both the  $x$ - and  $y$ -axis on a base-10 logarithmic scale. Ch MPI's latency is comparable to that of C MPI for cases (a) and (b) in the 64-bit Linux cluster setups. Ch MPI's latency is very close to that of C MPI for cases (c) through (e) on the heterogeneous and homogeneous network cluster setups in both Linux and Windows.

### Parallelized Execution Performance Testing

We ran a parallelized execution performance comparison between Ch MPI and MatlabMPI for the  $\pi$  calculation and an execution performance comparison between C MPI and Ch MPI for 2D matrix multiplication.

**Table 1. Execution times of non-MPI codes in a single processor.**

$n$	execution time (s)		
	<code>cpi_serial.exe</code>	<code>cpi_serial.c</code>	<code>cpi_serial.m</code>
4,800,000	0.0588	9.103	21.581
2,400,000	0.0295	4.55	10.772
1,600,000	0.0196	3.037	7.15

**Calculating  $\pi$ .** For this test, we used three machines from the Windows network cluster. We first evaluated the serial performances for non-MPI programs, `cpi_serial.c`, `cpi_serial.m`, and the compiled executable of `cpi_serial.c-cpi_serial.exe`. The serial performance provides execution times for the `for-loop` to calculate  $\pi$  with different intervals using a single processor. In Table 1,  $n$  represents the number of intervals. As Table 1 shows, the interpretive execution performance in Ch is approximately 2.35 times faster than Matlab in the serial performance test. Meanwhile, the execution of the compiled executable is approximately 154 times faster than the same C source code's interpretive execution.

To compare parallel execution performance, we used the programs `cpi_parallel.c`, `cpi_parallel.m`, and the compiled executable of `cpi_parallel.c-cpi_parallel.exe`. To create the program `cpi_parallel.c`, we slightly modified `cpi.c` in Figure 4 such that each process sums up the areas of consecutive rectangles instead of spaced rectangles. The program `cpi_parallel.m` is a Matlab program corresponding to `cpi_parallel.c`. MatlabMPI provides the MPI functions used in `cpi_parallel.m`.<sup>4</sup> In both programs, we set  $n$  to 4,800,000. We obtained the start time before all the processes began their computation, and the end time after the root process collected partial values from all other processes to obtain the final value of  $\pi$ . The execution time is the interval between the start and end times. Because the MPI function `MPI_wtime()` is not supported in MatlabMPI, we used Matlab functions `tic` and `toc` to obtain the execution time in `cpi_parallel.m`. To have a fair time evaluation against MatlabMPI, we used the equivalent standard C function `clock()` to get the execution time for running `cpi_parallel.c`. As Table 2 shows, the interpretive execution performance in Ch is approximately 2.35 times faster than Matlab in the parallel performance test. Meanwhile, the execution of the compiled executable is approximately 154 times faster than the same C source code's interpretive execution.



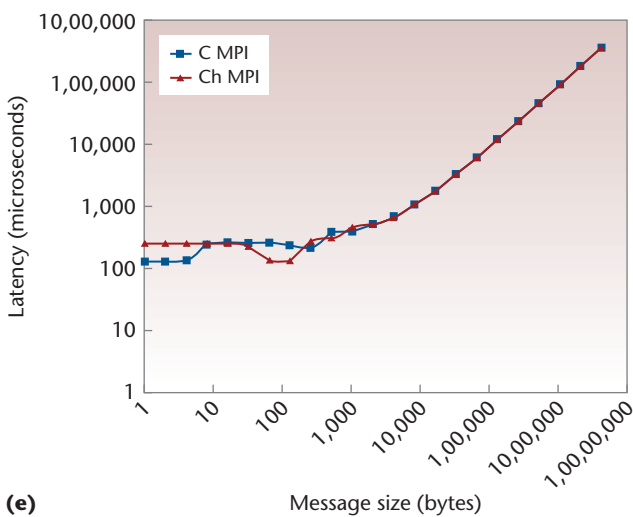
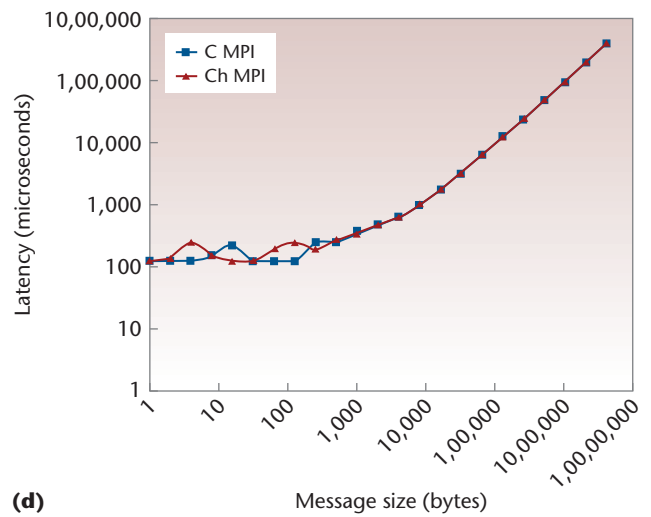
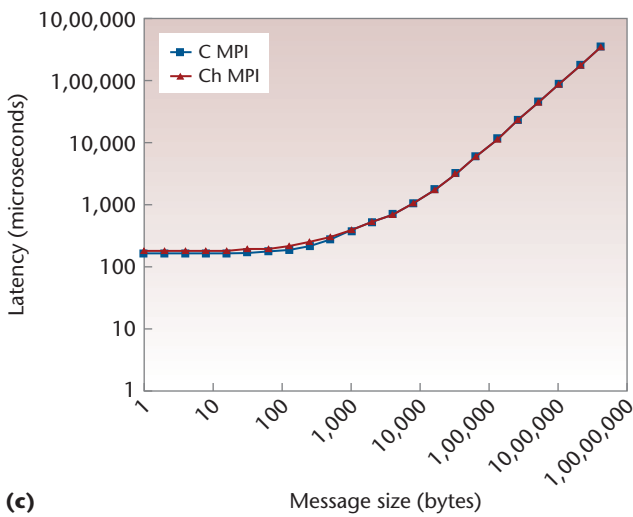
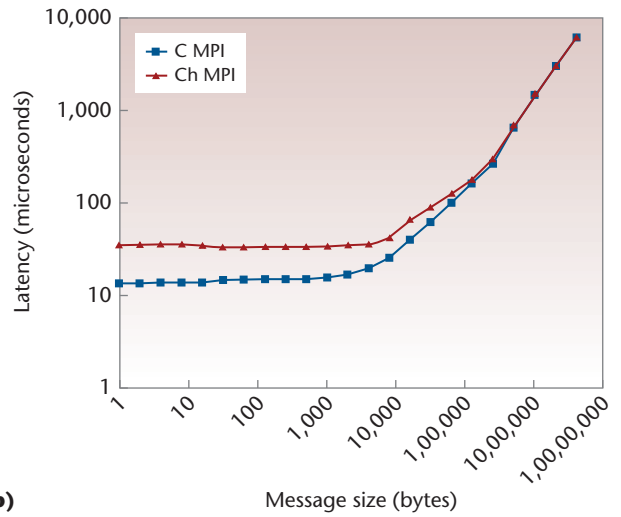
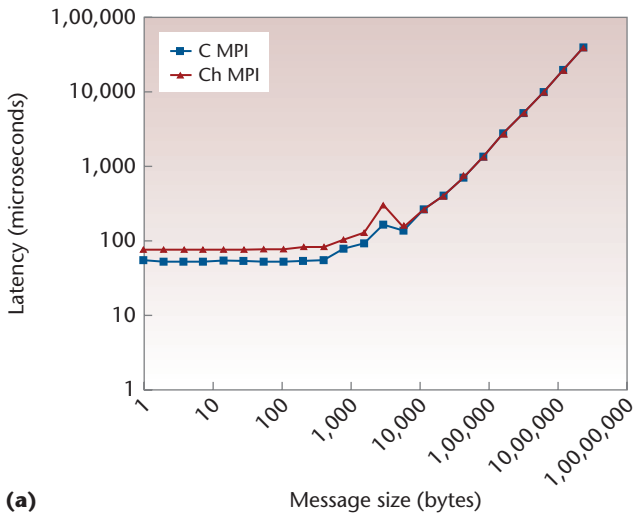


Figure 9. C MPI and Ch MPI latency test results in log-log plots. As in previous tests, the hardware setups were (a) a 64-bit Linux cluster with two processors and two nodes, (b) a 64-bit Linux cluster with two processors and one node, (c) a heterogeneous network cluster, (d) a Linux network cluster, and (e) a Windows network cluster.

Table 2. Execution performances of MPI codes.

# of processors ( $n = 4,800,000$ )	cpi_parallel.exe		cpi_parallel.c		cpi_parallel.m	
	execution time (s)	speedup	execution time (s)	speedup	execution time (s)	speedup
1	0.0598	1	9.206	1	21.597	1
2	0.03	1.993	4.609	1.997	10.85	1.991
3	0.02	2.99	3.068	3	7.25	2.979

As Tables 1 and 2 show, the programs spend most of their time on the `for-loop` that calculates  $\pi$ . As expected, executing the compiled executable is significantly faster than the corresponding interpretive execution of the same C source code or Matlab script. All methods scale linearly with the number of processors.

For a program that mainly performs the computation by an iteration statement, such as the `for-loop` that calculates  $\pi$ , it's significantly slower to run the program source code through an interpreter than a compiled executable. However, we can use Ch SDK to make binary functions, for some commonly used iteration statements, callable in a Ch script; with binary functions being called from the source code to perform time-consuming iterative calculations, the source code's execution performance can be comparable to that of a compiled executable.

**2D matrix multiplication.** A matrix multiplication `for-loop` performs time-consuming calculations. As an example, we used the program `mm_parallel.c` and its compiled executable `mm_parallel` to perform the second parallel execution test for matrix multiplication on the 64-bit Linux cluster, where each of the cluster's processors runs one process. We created a library, `libmm.a`, together with a header file, `mm.h`, to provide a function, `matrixmultiply()`, which performs 2D matrix multiplication. We generated the compiled executable `mm_parallel` with `libmm.a`. Using Ch SDK, we created a Ch dynamically loaded library, `libmm.dll`, and a Ch function file, `matrixmultiply.chf`, to make the function `matrixmultiply()` callable from within the Ch script `mm_parallel.c`.

We set the sizes of matrices A, B, and C in `mm_parallel.c` to  $2,880 \times 2,000$ ,  $2,000 \times 450$ , and  $2,880 \times 450$ , respectively. In this test, the master process didn't contribute to the computation, and we accounted only for the time spent by each worker process to complete its own computation. We determined the execution time by finding the maximum time among all the times spent by worker processes to finish the matrix multiplication

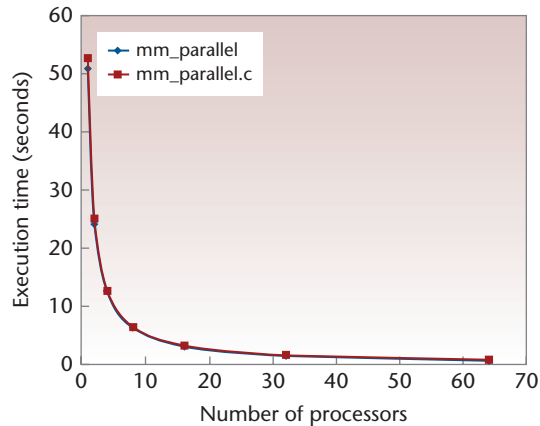


Figure 10. Execution times of `mm_parallel` and `mm_parallel.c`. Execution times of `mm_parallel.c` are almost the same as those of `mm_parallel`.

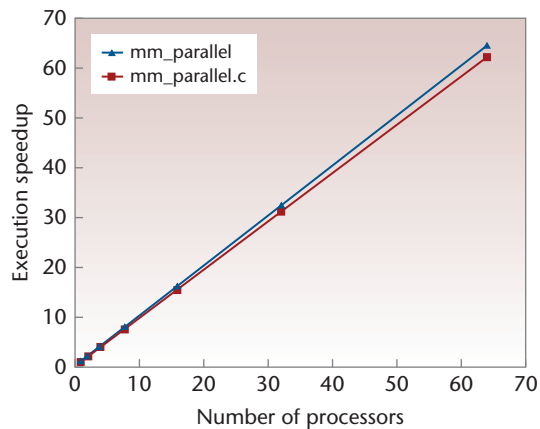


Figure 11. Execution speedups of `mm_parallel` and `mm_parallel.c`. Ch MPI scales linearly, with almost the same rate as C MPI.

function call. The MPI function, `MPI_Wtime()`, was called to obtain the time in seconds.

Figure 10 shows the execution times and Figure 11 shows the execution speedups from running these two programs with different numbers of processors. The number of processors in Figures 7 and 8 refers to the number of worker processes that contributed to the computation. As Figure 10 shows, the execution times of

```

#include <stdio.h>
#include <numeric.h>

int main()
{
    array double A[3][3] = {8,1,6,
                           3,5,7,
                           4,9,2};
    array double b[3][1] = {1,2,3};
    array double x[3][1];

    x = inverse(A)*b;
    printf("x =\n%7.2f", x);

    return 0;
}

```

Figure 12. Using Ch computational arrays to solve the equation  $\mathbf{Ax} = \mathbf{b}$ . With Ch computational arrays, we can specify matrix operations as scalar operations.

`mm_parallel.c` are almost identical to those of `mm_parallel`. Figure 11 shows that Ch MPI scales linearly, with almost the same rate as C MPI.

Because the function `matrixmultiply()` was called only once by each worker process, the `mm_parallel.c`'s execution times are almost identical to those of `mm_parallel`. As we noted earlier, if such a binary function is repeatedly called, a source code's execution time won't be comparable to that of a compiled executable. However, Ch MPI saves user development effort because it doesn't compile and link a program every time a change is made.

We can use Ch computational arrays to carry out matrix computation for applications that don't need to compile an MPI program for performance reasons with respect to other parts of the program. The program in Figure 12 shows how to use Ch computational arrays to solve the equation  $\mathbf{Ax} = \mathbf{b}$ , given that

$$\mathbf{A} = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}.$$

The header file `array.h` must be included to utilize Ch computational arrays. As Figure 12 shows, the program's header file `numeric.h` contains the header file `array.h` and the function prototype of the function `inverse()` for computing the inverse of a matrix. Matrix `A` and vectors `b` and `x` are declared as Ch computational arrays of type `array double`. We obtain the solution vector `x` simply through

the statement `x = inverse(A)*b`. The program's output is

```

x =
0.05
0.30
0.05

```

With Ch computational arrays, we can specify vector and matrix operations in a C source code as scalar operations, as Figure 12 shows. In addition, declaring a matrix as a Ch computational array is straightforward regardless of the matrix's size.

### Applications

We now describe two example Ch MPI applications. The first involves task farming with existing precompiled executables and the second involves Web-based parallel computing in a heterogeneous environment.

#### Task Farming with Ch MPI

Task farming is a technique used to achieve high throughput of serial tasks on parallel computers. MPI is a suitable approach for implementing task farm programs that benefit from parallel computing.<sup>9</sup> A task farm program essentially consists of a master section and a worker section. When the task farm program is running in a parallel environment, a single master process and several worker processes are launched to perform whatever is defined in the program's master and worker sections. Each process runs on a different processor.

In general, the master process receives requests from worker processes for tasks and sends tasks to worker processes until all tasks have been finished; it then notifies worker processes to quit. Until notified to quit, worker processes continuously request and execute new tasks from the master process. In the task farm paradigm, tasks are independent of each other and usually represent an intensive computation to be performed on a worker processor. Therefore, executing independent tasks usually means calling/running a common precompiled function/executable with different inputs.

Some task farm programs are for applications that must run an existing precompiled executable over a set of parameters to generate a set of results—such as in parameter searching or ensemble studies. Such programs are mainly composed only of functions dealing with message transfer between the master and worker processes. In addition, the master part of those task farm programs can also change the conventional first-come,

first-serve strategy by adopting different algorithms to coordinate the task distribution among worker processes for different concerns. In some cases, for example, the master process can specify the number of tasks sent to a particular worker process each time or limit the total amount of tasks performed by certain worker processes. Ch MPI can effectively facilitate the design of such a task farm program because a user can change the program and readily test it without the compilation process.

As an example, a simple MPI task farm program `task_farm.c` can run on different numbers of processors to handle a total of 64 identical tasks on our Linux cluster. The tasks are saved in an input file, `tasks`, for the task farm program. Each task consists of consecutive commands `cd output; ./nbody 1000 > out_1`. Thus, each task stands for changing to a subdirectory, `output`, running an executable, `nbody`, from the parent directory with a command line argument, `1000`, and redirecting the executable's output to a file, `out_1`. All the commands for one task should be separated by semicolons and on the same line. Because each task is identical, a linear execution speedup is expected as the number of processors increases.

The `nbody` executable performs a simplified  $N$ -body simulation<sup>10</sup> associated with a 1D line of  $N$  particles. The number of particles is specified as a command line argument. Each particle has an initial position on the  $x$ -axis. Evaluating a particle's position depends on all the forces exerted on this particle from all other particles. A force exerted on a particle from another particle is proportional to the inverse of the square of the distance between these two particles. Forces on a given particle from any particles on its left side have a negative value because they're pulling the particle in the negative  $x$  direction; forces from any particles on its right side have a positive value. The program eventually outputs the average distance between two adjacent particles.

The execution time in `task_farm.c` is the time required to finish all 64 tasks. Figure 13 shows execution speedups for different numbers of processors running `task_farm.c`; a linear speedup is obtained as the number of processors increases.

With Ch MPI, users can readily run a task farm program such as `task_farm.c` on multiple processors without compilation to decrease task-completion time.

### Web-Based Application of Ch MPI for Parallel Computing

The CGI is a standard for interfacing external applications with Web servers. A Web server

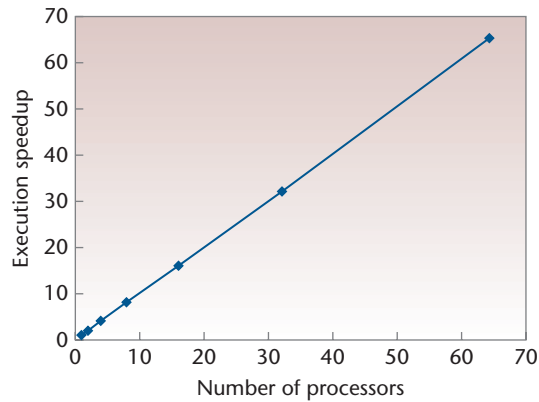


Figure 13. Execution speedups for different numbers of processors running `task_farm.c` to finish 64 identical tasks. A linear speedup is obtained as the number of processors increases.

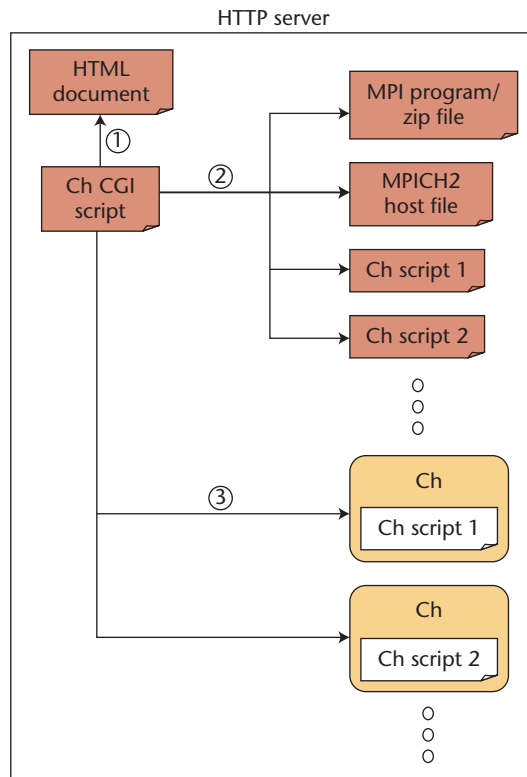


Figure 14. A Ch CGI script performs multiple tasks on the Web server in the Web-based parallel computing application. It collects information from an HTML document, retrieves an uploaded MPI program or archived file, creates an MPICH2 host file and other Ch scripts, and invokes Ch interpreters to run those scripts.

can execute a CGI program to run external applications and output dynamic results. Although developers can write a CGI program in any language, using scripting languages makes it easier to debug, modify, and maintain.

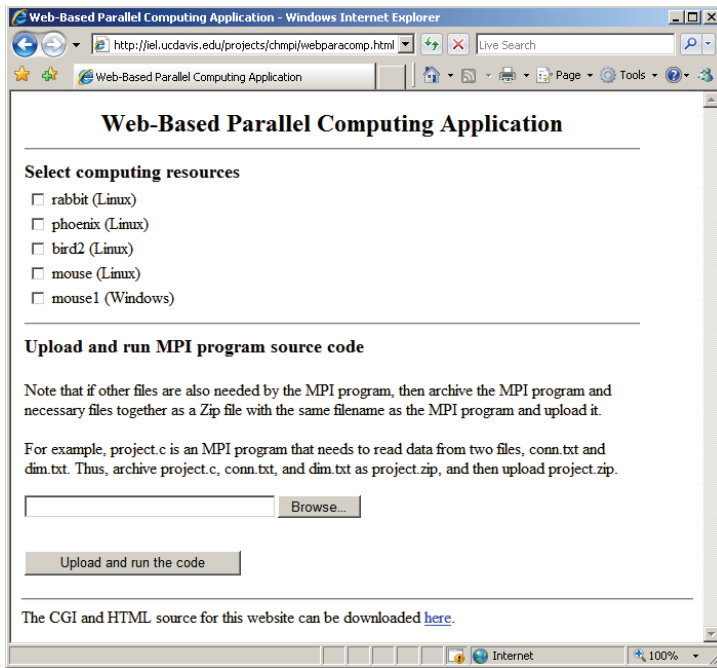


Figure 15. The user interface of the Web-based parallel computing application. Through the user interface, a user can select the computing resources, upload an MPI program source code or an archived file, and view the execution results.

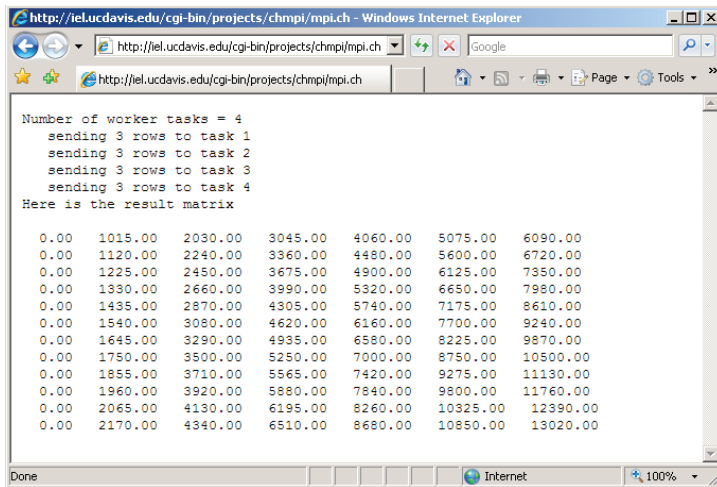


Figure 16. Results from running `mm_parallel.c` through the Web-based parallel computing application. The resultant matrix is obtained by parallel 2D matrix multiplication on four computers.

Ch is suitable for Web-based applications due to its interpretive nature. Like Java Server Page and Active Server Page, the Ch CGI Toolkit provides four easy-to-use classes for CGI programming in Ch: `CResponse`, `CRequest`, `CServer`, and `CCookie`. We've developed many Web-based applications—such as a Web-based system for control system design and analysis—using the Ch CGI Toolkit.

In our example case here, we use the Apache HTTP server as the Web server for our parallel computing system. Users' requests are sent to the Web server as HTML documents. The computing resources consist of our Linux and Windows network clusters.

As Figure 14 shows, a Ch CGI script in the Web server performs multiple tasks. First, the Ch CGI script parses an HTML document to collect useful information. Next, it retrieves from the collected information an uploaded MPI program or archived file, generates an MPICH2 host file, and further creates several Ch scripts. Finally, the Ch CGI script invokes Ch interpreters to run the created Ch scripts. Each created Ch script performs one of the following tasks:

- distribute the MPI program or archived file to the machines specified in the generated MPICH2 host file;
- start the SMPDs;
- extract files from the archived file if needed;
- execute the MPI program on the specified hosts;
- remove the uploaded file, and extracted files if any, from the specified hosts; and
- terminate the SMPDs.

Figure 15 shows our Web-based parallel computing system's user interface. Users can select the computing resources to use and upload a C MPI program source code for execution. For some specific MPI programs, such as computational fluid dynamics simulations, extra files like a grid file and an initial condition file are required. In such cases, a user can upload an archived file that contains a C MPI program source code along with all the necessary files through the user interface. Figure 16 shows the results from running `mm_parallel.c` on four computers with small matrices through the Web-based parallel computing application. The sizes of the matrices **A**, **B**, and **C** are  $12 \times 15$ ,  $15 \times 7$ , and  $12 \times 7$ , respectively.

**A** Ch MPI package lets developers run identical MPI C scripts across heterogeneous platforms. Interpretive computing in Ch MPI is efficient in comparison with other scripting solutions, running more than twice as fast as both Matlab (for sequential  $\pi$  calculation) and MatlabMPI (for parallel  $\pi$  calculation). Also,

using Mobile-C ([www.mobilec.org](http://www.mobilec.org)), users can dynamically generate, deploy, and execute C programs written with Ch MPI across different platforms in a network. Such dynamic parallel computing has great potential for many challenging applications such as data mining, autonomic parallel computing, grid computing, and cloud computing.

All of the codes we described in this article are available on the Ch MPI Web site (<http://iel.ucdavis.edu/projects/chmpi>).

## References


1. M. Snir et al., *MPI: The Complete Reference—The MPI Core*, 2nd ed., MIT Press, 1998.
2. L. Dalcin, R. Paz, and M. Storti, "MPI for Python," *J. Parallel and Distributed Computing*, vol. 65, no. 9, 2005, pp. 1108–1115.
3. L. Dalcin et al., "MPI for Python: Performance Improvements and MPI-2 Extensions," *J. Parallel and Distributed Computing*, vol. 68, no. 5, 2008, pp. 655–662.
4. K. Jeremy and S. Ahalt, "MatlabMPI," *J. Parallel and Distributed Computing*, vol. 64, no. 8, 2004, pp. 997–1005.
5. E. Ong, "MPI Ruby: Scripting in a Parallel Environment," *Computing in Science & Eng.*, vol. 4, no. 4, 2002, pp. 78–82.
6. H.H. Cheng, "Scientific Computing in the Ch Programming Language," *Scientific Programming*, vol. 2, no. 3, 1993, pp. 49–75.
7. H.H. Cheng, *C for Engineers and Scientists: An Interpretive Approach*, McGraw-Hill, 2009.
8. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with Message Passing Interface*, 2nd ed., MIT Press, 1999.
9. P. Marques, "Task Farming and the Message Passing Interface," *Dr. Dobbs's J.*, vol. 28, no. 9, 2003, pp. 32–37.
10. S.J. Aarseth, *Gravitational N-Body Simulations: Tools and Algorithms*, Cambridge Univ. Press, 2003.

**Yu-Cheng Chou** is an assistant professor in the Department of Mechanical Engineering at Chung Yuan Christian University, Taiwan. His research interests include mobile agent-based computing, autonomic computing, parallel computing, intelligent mechatronics and embedded systems, and software and system integration. Chou has a PhD in mechanical and aeronautical engineering from the University of California at Davis. He is a member of the IEEE and the ASME. Contact him at [ycchou@cycu.edu.tw](mailto:ycchou@cycu.edu.tw).

**Stephen S. Nestinger** is an assistant professor in mechanical engineering at the Worcester Polytechnic

Institute. His research interests include cooperative multirobot systems, robotics, space robotics, vision systems, control systems, mechatronics, real-time systems, software and systems integration, and mobile agent systems. Nestinger has a PhD in mechanical and aeronautical engineering from the University of California at Davis. He is a member of the IEEE and the ASME. Contact him at [ssnestinger@wpi.edu](mailto:ssnestinger@wpi.edu).

**Harry H. Cheng** is a professor in the Department of Mechanical and Aerospace Engineering and the Graduate Group in Computer Science at the University of California, Davis, where he directs the Integration Engineering Laboratory. He is the founder of SoftIntegration, Inc., and the original designer and implementer of an embeddable C/C++ interpreter Ch for cross-platform scripting. His research is focused on computer-aided engineering, mobile-agent-based computing, intelligent mechatronic and embedded systems, and innovative teaching. He is author of *C for Engineers and Scientists: An Interpretive Approach* (McGraw-Hill, 2009). Cheng has a PhD in mechanical engineering from the University of Illinois at Chicago. He is a fellow of the ASME and a senior member of the IEEE. Contact him at [hhcheng@ucdavis.edu](mailto:hhcheng@ucdavis.edu).

 Selected articles and columns from IEEE Computer Society publications are also available for free at <http://ComputingNow.computer.org>.



**Call for Articles**

IEEE Software seeks practical, readable articles that will appeal to experts and nonexperts alike. The magazine aims to deliver reliable information to software developers and managers to help them stay on top of rapid technology change. Submissions must be original and no more than 5,400 words, including 200 words for each table and figure.

Author guidelines: [www.computer.org/software/author.htm](http://www.computer.org/software/author.htm)  
 Further details: [software@computer.org](mailto:software@computer.org)  
[www.computer.org/software](http://www.computer.org/software)

**IEEE Software**